Developer Technical Support

Apple IIGS
#80: QuickDraw II Clipping

Written by: Eric Soldan March 1990

This Technical Note explains a lot about QuickDraw II operation, specifically clipping.

Before Beginning

Before beginning this Note, some statements, disclaimers, and definitions:

1. This is not a substitute for the QuickDraw II introduction in the *Apple IIGS Toolbox Reference*, but rather a supplement.
2. A pixelmap is a series of bytes that hold pixel data whose rectangular shape is defined by a `LocInfo` structure.

This Note describes in great detail the way that QuickDraw II does things with pixelmaps. It begins with a description of the `LocInfo` structure, which is the most important thing to understand in terms of QuickDraw II pixelmap management. Once this is understood, this Note covers how it applies to using functions such as `PPToPort`, `PaintPixels`, and `CopyPixels`. And once this is understood, it then describes how `LocInfo` structures are used to control drawing into a `grafPort`. (`PPToPort` is used in this Note. `PaintPixels` and `CopyPixels` are very close in function to `PaintPixels`. The information and theory in this Note also apply to these calls.)

Understanding the material in this Note should help you better understand the entire toolbox. It is surprising how much can be accomplished with the toolbox without completely understanding these concepts; it is also surprising how much easier programming with the toolbox gets when these concepts are fully understood.

Note: Structures are written with C syntax in this Note. In addition, this Note uses the screen address 0xE12000L. The possibility of shadowing being active and the screen address being 0x12000L is ignored.

The Beginning

One must begin with the `LocInfo` structure, which is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          width;                /* Width */
       Rect          boundsRect;           /* BoundsRect */
};
```

For this Note, one can change this structure a little bit by calling the `width` element `rowBytes`. This convention is good because `rowBytes` is more descriptive than `width` (it indicates that one is measuring the width in bytes) and it allows one to use the word "width" elsewhere in this Note without confusion. So for the purposes of the Note, the new LocInfo structure definition is as follows:

```
struct LocInfo {
       Word          portSCB;              /* SCB in low byte */
       Pointer       ptrToPixImage;        /* ImageRef */
       Word          rowBytes;             /* Width in bytes*/
       Rect          boundsRect;           /* BoundsRect */
};
```

The `ptrToPixImage` field is a pointer to some block of bytes in memory. (This block of bytes is referred to as the `pixImage` from here on.) A `pixImage` doesn't have any inherent shape. QuickDraw II deals with it as a rectangle, and the `LocInfo` record defines the rectangularity of it.

When saving a 32,000 byte screen image, one doesn't save the number of bytes of which each row consists. One assumes that each row is 160 bytes by convention, and this is a safe assumption, since the IIGS video hardware expects 160 bytes. But the point is that in the 32,000 bytes of screen data, there is no indicator as to the specific size of a row. One must just know that it is 160 bytes per row. This size is fine for screen shots, but it is not fine when different pixelmaps can be different widths. If they can be different widths, then one also needs some information as to what those widths are, hence the `portSCB`, `rowBytes`, and `boundsRect` fields in a `LocInfo` structure.

The `boundsRect` and `portSCB` fields tell the shape of the pixelmap in pixels, the `boundsRect` tells how many pixels wide and tall the pixelmap is, and the `portSCB` tells how big those pixels are (320-mode pixels are four bits wide and 640-mode pixels are two bits wide). One would think that this would be enough information to determine the size of the `pixImage`, but it isn't. The `rowBytes` can be larger than the `boundsRect/portSCB` would indicate (see Figure 1). This situation is legal; it means that some bytes are being wasted, but it is legal.

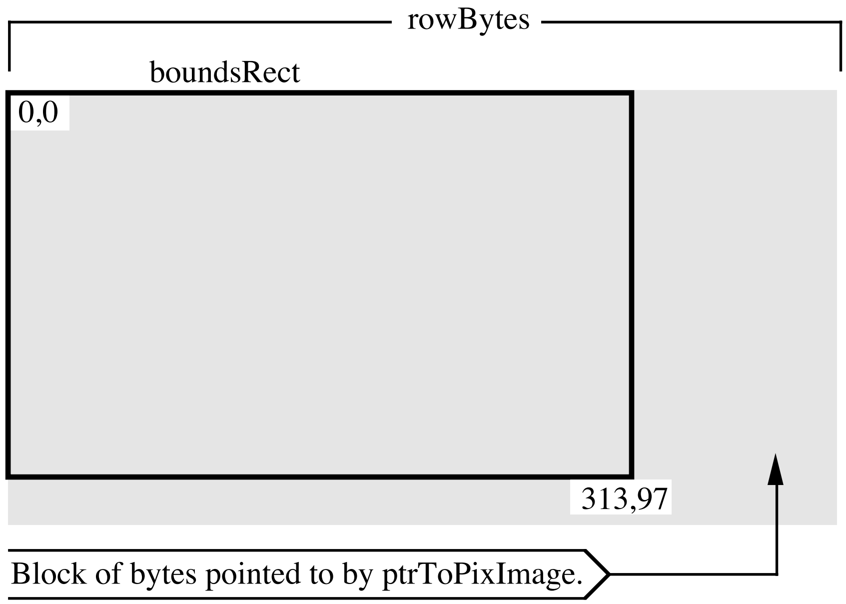

Figure 1–Sample LocInfo Structure

One simply has to know the size of the `pixImage`, since it cannot be determined by the `LocInfo` information. If the `pixImage` is the screen, then it is 32,000 bytes. If it is a fixed or locked handle, then one can do a `FindHandle` on the pointer followed by a `GetHandleSize` on the found handle.

Figure 1 represents a sample `LocInfo` structure. The `portSCB` (although not pictured) is also relevant, as it determines the size of the pixels. If the pixelmap is a 320-mode pixelmap, one could change it to a 640-mode pixelmap by changing the `portSCB` to 640 mode and doubling the width of the `boundsRect`. In doing this conversion, note that `rowBytes` is not affected and that the `pixImage` does not change size.

In the example illustrated in Figure 1, the `pixImage` is bigger than the `boundsRect`, but again, this is okay. However, this is not the case for the screen, where the `rowBytes` is 160 and the height of the `boundsRect` is 200 (the size of the screen is exactly equal to 160 * 200 = 32,000).

There are some rules to determining the `rowBytes` value. First, `rowBytes` must not be too small. This is obvious. Second, `rowBytes` must be evenly divisible by eight. This is not at all obvious, but it is very important. QuickDraw II makes some assumptions for speed, and one of them is that `rowBytes` is a multiple of eight.

So much for describing the `LocInfo` structure. Now for how to use it via `PPToPort`.

`PPToPort` accepts (among other things) a pointer to a source `LocInfo` record and a pointer to a source rectangle. `PPToPort` does not use the source rectangle directly; it first intersects it with the `boundsRect` in the `LocInfo` record, and it uses this intersection rectangle instead. This intersection rectangle guarantees that the area involved is completely enclosed by the

boundsRect (and therefore within the pixImage). If the source rectangle is entirely outside the boundsRect, then the intersection of the source rectangle and the boundsRect is empty, thus nothing is drawn.

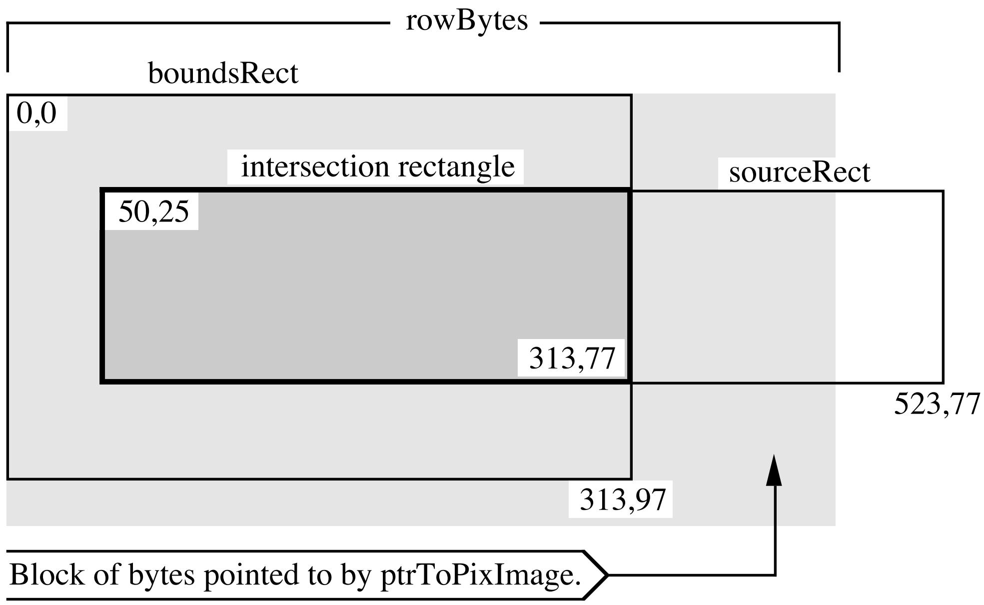

Figure 2–Sample LocInfo Structure With sourceRect

Figure 2 contains a sourceRect which is not completely contained by the boundsRect; the sourceRect is so wide that it even goes beyond the edge of the pixImage. If the entire contents of this rectangle were drawn, the result would be quite a mess, since it extends beyond the boundary of the pixelmap. However, PPToPort first intersects the sourceRect and the boundsRect, and then uses the resulting intersection rectangle (illustrated with a thicker border in the figure). PPToPort uses only the contents of the intersection rectangle.

Up until now, the boundsRect upper-left corner has always been 0,0. This is an easy way to think of it, but it is not necessary. The important thing to remember about these rectangles is their relation to one another. If one were to offset both the boundsRect and sourceRect in this example, the values for the corners of the rectangles would change, but the relationship between the two rectangles would stay the same. Figure 3 illustrates the same example if one were to offset both rectangles by -60,-45.

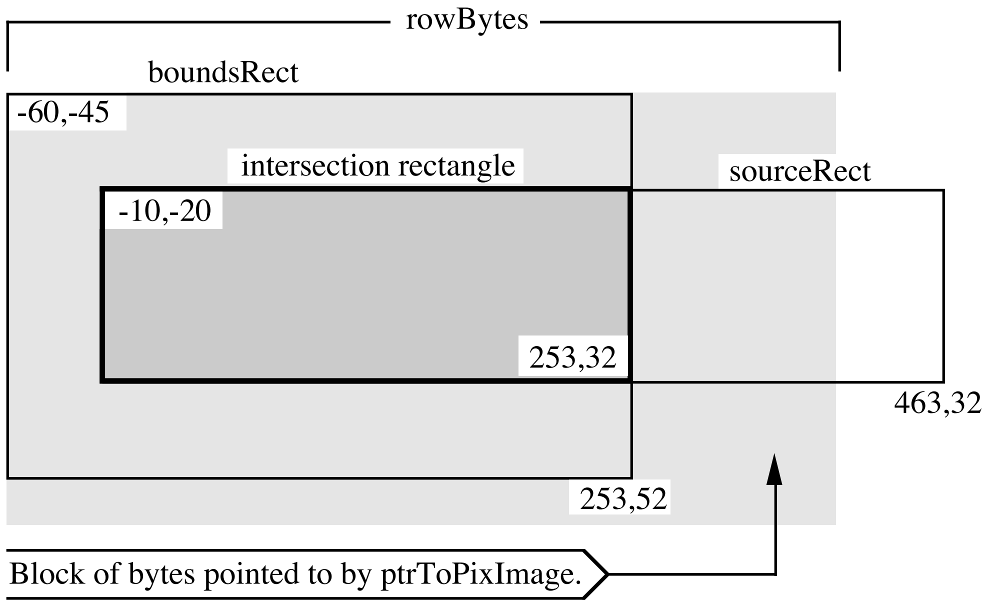

Figure 3–Sample LocInfo Structure Offset by -60,-45

Notice that the same area of the `pixImage` is involved, even though the `boundsRect` and `sourceRect` are offset. When one offsets both the `boundsRect` and `sourceRect` by the same amount, the referenced part of the `pixImage` does not change—this is an important concept.

Time to ask a question that is answered shortly: "Why isn't the upper-left corner of the `boundsRect` always 0,0?" Because the `LocInfo` record isn't always a source `LocInfo` record. It can also be a destination `LocInfo` record, and the most common pixelmap to which a destination `LocInfo` record refers is the screen.

If you had not noticed, the discussion changes gears here—to discuss `LocInfo` records that indicate a destination pixelmap. Basically, everything is the same as has been described with two exceptions. First, destination pixelmaps do not have a `sourceRect`. Instead there is a rectangle that describes some portion of the destination pixelmap, and this rectangle is called the `portRect`. Second, the `LocInfo` record is part of a `grafPort`, and each `grafPort` has a `LocInfo` record as part of the `grafPort` data structure.

It is important to remember that a `LocInfo` record can be used as either a source or destination `LocInfo`. All a `LocInfo` record does is define some bytes in memory as a `pixImage`. Even the screen, which is usually used as a destination pixelmap, can be used as a source pixelmap. There could be situations where one might want to take part of the screen and copy it into some off-screen pixelmap, and in this case, the screen would be a source of pixel data, not a destination.

In the case of the screen pixelmap, there are no wasted bytes in the `pixImage`, as all of the screen bytes are enclosed by the `boundsRect`. The screen width of 160 is evenly divisible by

eight, so there is no slop at the right edge, and there are no extra rows hanging off the bottom of the `boundsRect`.

Figure 4 shows a sample `LocInfo` and `portRect` (every `grafPort` has a `LocInfo` and a `portRect`).

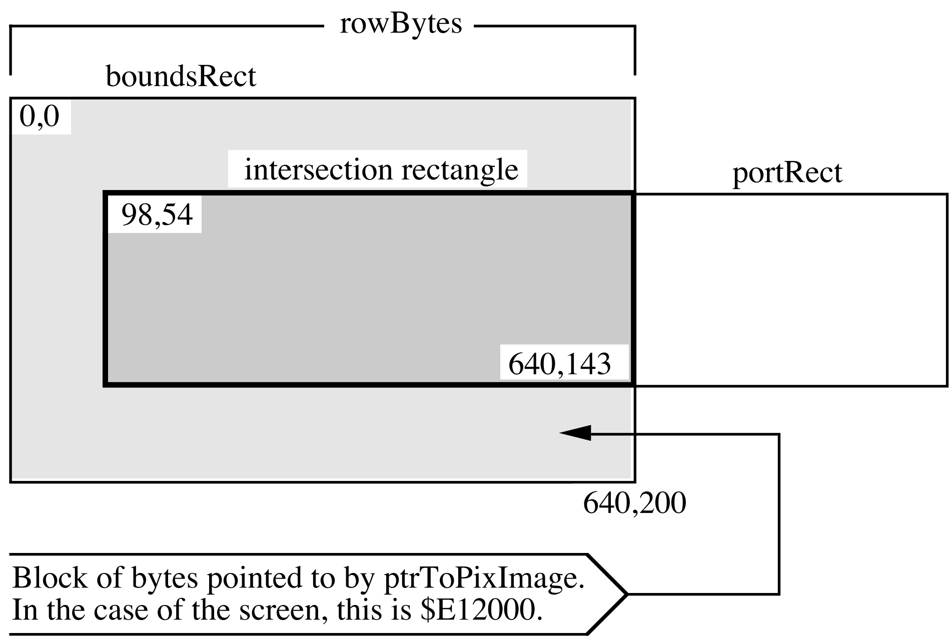

Figure 4–Sample LocInfo and portRect

Following are two important points to remember:

1. Every `grafPort` works in local (not global) coordinates (local coordinates are defined soon).
2. The origin of the `grafPort` is the upper-left corner of the `portRect`. There is no `GetOrigin` call; there is a `SetOrigin` call, but no `GetOrigin`. To get the origin of a `grafPort`, one needs to do a `GetPortRect` call, and then look at the upper-left corner to determine the current origin of the `grafPort`. This is **the** way to get the origin.

In the case of Figure 4, local and global coordinate systems are the same, as is always the case when the `boundsRect` has an upper-left corner of 0,0 (which it seldom does). So, for this exceptional case, one doesn't need a definition of local coordinates. In the global coordinate system, the upper-left corner of the screen is 0,0. In local coordinates, the upper-left corner of the screen is whatever the `boundsRect` says it is. So when the upper-left corner of the `boundsRect` is 0,0, the global and local coordinate systems are the same.

In Figure 4, if one tried to draw something to point 0,0, it would not draw—it would be clipped because it is outside the portRect. So even if one tried to draw there, it would not change point 0,0. If a user moved a mouse to that location and an application performed a `GetMouse` (which

returns the mouse location in the local coordinates of the current `grafPort`), it would return 0,0 as the mouse location.

If one did a `SetOrigin(0,0)`, then the `boundsRect` and `portRect` would be offset by the difference between the old and new origins. Both rectangles would be offset, so the relationship between them would remain the same, as Figure 5 illustrates.

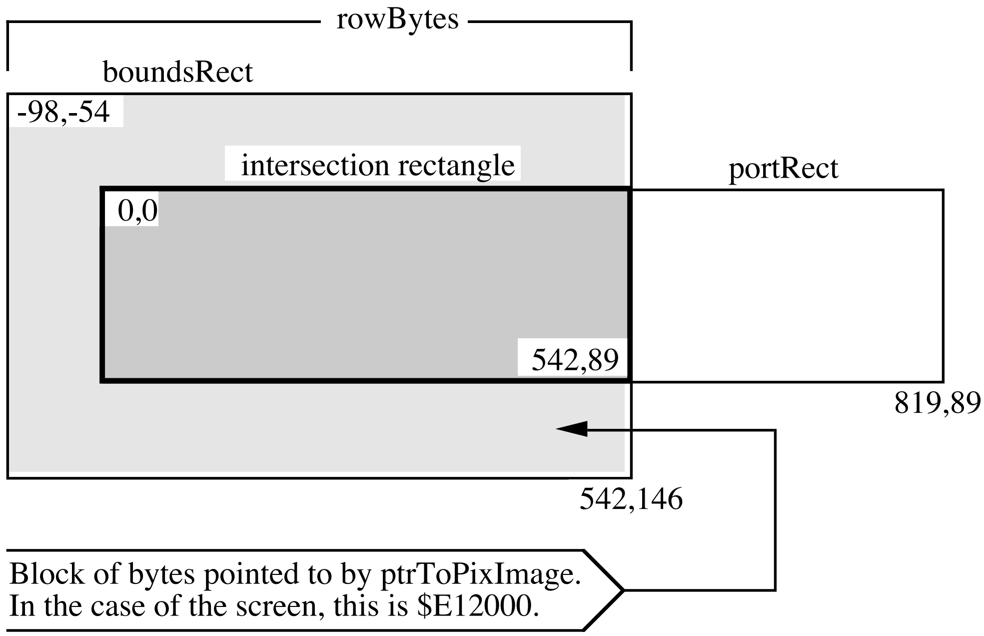

Figure 5–Sample LocInfo and portRect, Both Offset

Now if a user moves a mouse to the upper-left corner of the screen, a call to `GetMouse` returns a value of -98,-54, as expected, and if a user moves the mouse to the upper-left corner of the `portRect`, a call to `GetMouse` returns 0,0, again as expected. This is how origins work and how the conceptual drawing space relates to the `grafPort`. The `boundsRect` of the `grafPort` (in the `LocInfo` record of the `grafPort`) and the `portRect` of the `grafPort` are offset when one calls `SetOrigin`. It is that simple.

Now that it is simple, time to complicate matters with one more player in the QuickDraw II clipping world: the `visRgn`.

The `visRgn` exists for one purpose: to cause more clipping. It never causes anything to be clipped less than the `portRect` does, and in the case of a top window that is completely visible, the `visRgn` and the `portRect` are exactly the same size. Even more than that, the enclosing rectangle for the `visRgn` (every region has an enclosing rectangle) is this case would be exactly the same as that of the `portRect`. This all makes sense when one looks at the purpose of a `visRgn`. Again, the `visRgn` can only cause more clipping. If the entire window is visible, one does not want more clipping, so a `visRgn` the same size as the `portRect` guarantees that it does not clip any more than the `portRect`, as it must clip the same amount.

The `visRgn` is a different size than the `portRect` when the window is not the top window and part of it is overlapped (or if part of the window is off the screen). The part that is overlapped is excluded from the `visRgn`, and this excluded part is clipped to protect the window above from being drawn upon. This is how window clipping works. This is all there is to it.

Figure 6 enhances Figure 5 by adding an overlapping window to demonstrate the `visRgn`.

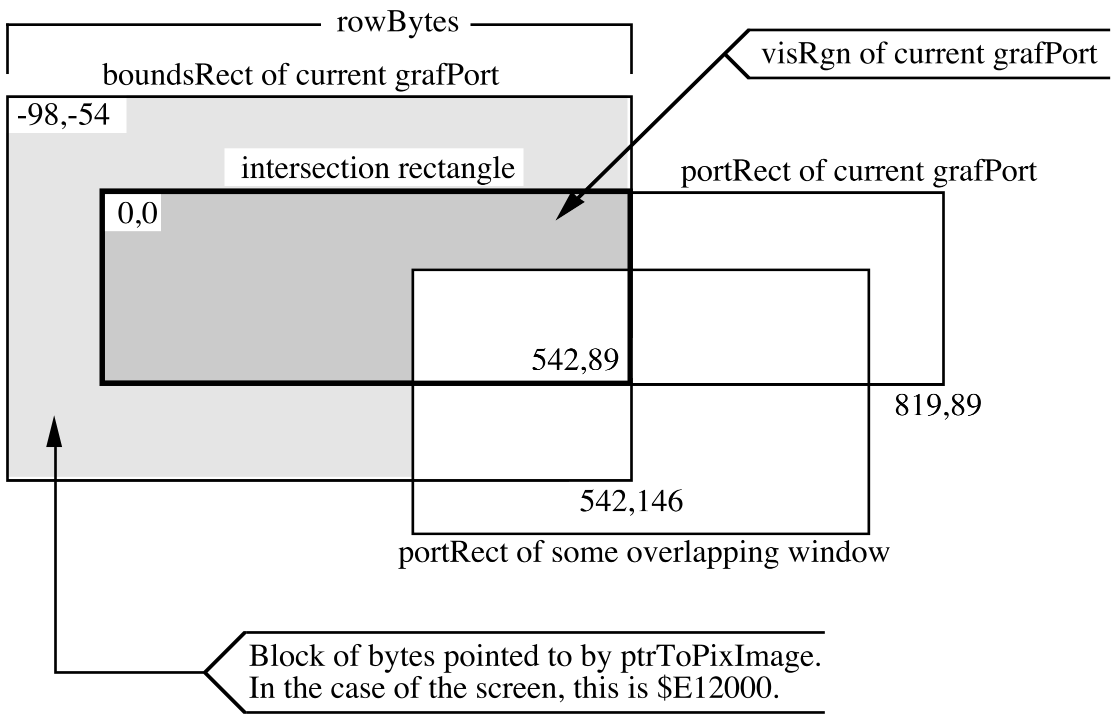

Figure 6–Sample LocInfo and portRect With Overlapping Window

What happens to the `visRgn` during a `SetOrigin`? Remember that the `boundsRect` and `portRect` get offset. The `visRgn` does too. Again, if all of these elements are offset together, then the relationship between them remains the same; they stay the same, relative to one another. (For more information, see Einstein's theory of general relativity.)

The final component for clipping is the `clipRgn`, which is the application's property and, therefore, the application's responsibility. The system sets the `clipRgn` about as big as it can get to start (much bigger than the `portRect`); this is often referred to as arbitrarily large, even though it isn't so arbitrary. The system creates all `grafPort` structures with a large `clipRgn`, and this can be a problem for certain types of QuickDraw II operations. Since the `clipRgn` already reaches to the borders of the conceptual drawing space, it cannot be offset; it is effectively stuck, due to its size. It is a good practice to make the `clipRgn` smaller than the system default.

`SetOrigin` does **not** offset the `clipRgn`. (This is why the size problem with a big `clipRgn` is not so apparent.) The `clipRgn` is the only clipping component that is not offset by

SetOrigin, and one should consider this when using clipRgn for clipping effects, since an application must remember to offset it if it needs to be offset.

Now with all of the fundamentals out of the way, it is time to play some grafPort clipping games. As a refresher, there are four clipping components in a grafPort: the boundsRect, the portRect, the visRgn, and the clipRgn.

If an application creates its own off-screen grafPort structures, then it can do as it wishes with all four clipping components. After all, if it has the responsibility to set them up in the first place, it should have the right to change them. If, however, the Window Manager creates the grafPort structures, then an application should keeps its figurative hands off certain clipping components, namely the boundsRect and the visRgn. The clipRgn, by definition, is the application's to do with as it sees fit, and if careful, an application can also change the portRect. Changing the portRect can be very useful, but one needs to be careful and fully understand all of the ramifications.

So, why would one change the portRect, and how would one do it?

Another figure is in order.

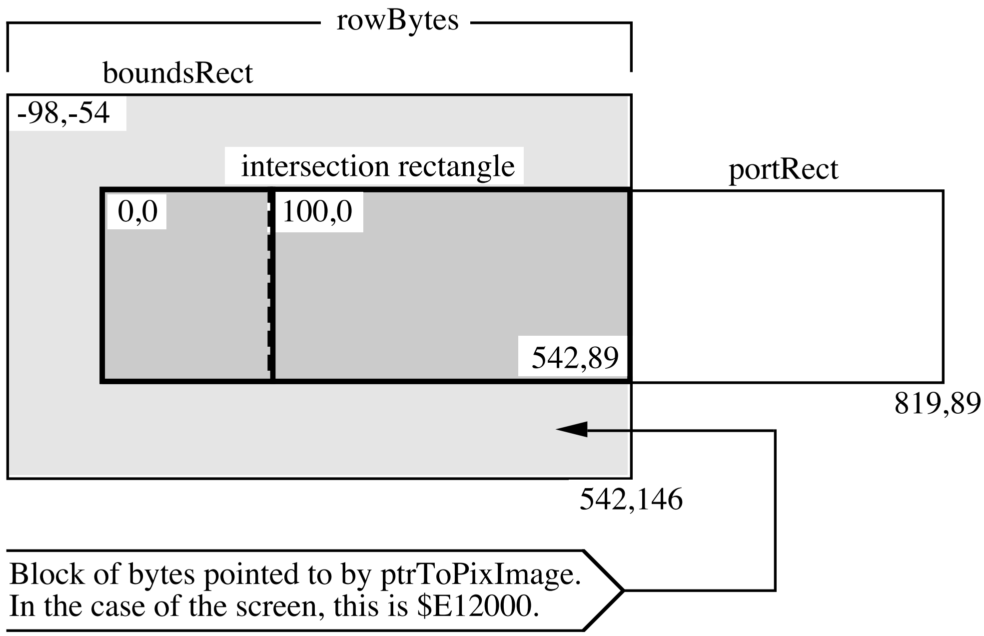

Figure 7–Sample LocInfo and Modified portRect

One can use the GetPortRect call to get the portRect for the current grafPort. One can then modify it, and then use the SetPortRect call to inform the grafPort about the change. Why do this? In Figure 7, the dotted line represents the new left edge of the portRect after the modification (a simple modification of adding 100 to the old value of zero).

Note that changing the `portRect` in this way changes the relationship between the `portRect` and the `boundsRect`. Anything drawn from 0 to 99 (x coordinate) is clipped, since it is outside the new (modified) `portRect`. Before the modification, anything drawn from 0 to 99 would have affected the screen.

This modification may cause the `portRect` to be smaller than the `visRgn`. This is okay, since the `visRgn` can only cause more clipping, not less. So, all of this works just fine. Note that the origin changed when the left edge of the `portRect` changed. The upper-left corner of the `portRect` is always the origin, and an application changed it. The origin changed without a `SetOrigin` call. (Scary, huh?)

One could have done exactly the same thing by making a `clipRgn` to exclude the x coordinates from 0 to 99. However, here is something cool. After the modification, do a `SetOrigin(0,0)`, which sets the upper-left corner of the shrunk `portRect` to 0,0. One cannot accomplish this sort of thing as simply by making a `clipRgn`. One can effectively move where an origin of 0,0 is the screen, and just building a `clipRgn` to exclude some part of the screen does not accomplish this.

Why would one want to change where 0,0 is on the screen? This sort of trick is very useful for adding rulers to a document window, for example. One of the problems with rulers is that they should not scroll with the rest of a document. Unfortunately, `TaskMaster`, if allowed to handle scrolling, doesn't know about a ruler at the top of a window and scrolls it with the rest of the window's content area. By changing the `portRect` so that the ruler is not inside of it, one can keep `TaskMaster` from scrolling it. In a draw procedure, when it is necessary to draw the ruler, grow the `portRect`, set the origin to 0,0, and then draw the ruler. Once it is drawn, set the `portRect` back to the smaller size to protect the ruler again.

Another reason one might want to do this is if an application uses a split window (where the top of the window may show a different part of the document than the bottom). Changing the `portRect` has the advantage that the upper-left corner of the `portRect` is always the origin, so it makes mapping document coordinates easier.

Another advantage to using the `portRect` in this way is that it keeps the `clipRgn` free for other purposes. Being able to separate types of clipping to either the `portRect` or the `clipRgn` keeps the `clipRgn` from being overused.

As a final note, it should be observed that the only clipping that is done is on a destination pixelmap. There is no clipping on a source pixelmap. There is no need. All the clipping needed is done at the destination end, so it would be wasteful to clip twice.

This finishes the discussion about QuickDraw II and how the `boundsRect`, `portRect`, `visRgn`, and `clipRgn` work together to accomplish clipping. Hopefully this Note answers more questions than it creates.

Further Reference

- *Apple IIGS Toolbox Reference,* Volume 2

- *Relativity the Special and General Theory* (1920)